Distributed Big Data Library
Apache Spark

LEARNING, APPLYING, MULTIPLYING BIG DATA ANALYTICS

This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under grant agreement No 809965.

Shortcomings of Mapreduce

Solution?

- New framework: Support the same features of MapReduce and many more.
- Capable of reusing Hadoop ecosystem : e.g HDFS, YARN, etc.

- Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

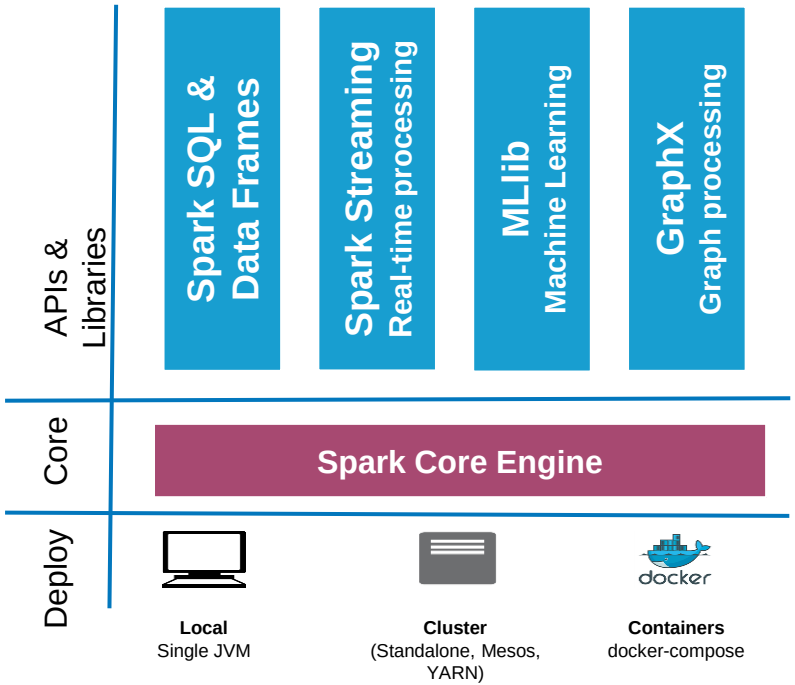
Apache Spark

Introduction to Spark

- Open source
- Distributed
- Scalable
- In-memory
- General-purpose
  - High level APIs
    - Java
    - Scala
    - Python
    - R
  - Libraries
    - MLlib
    - Spark SQL
    - GraphX
    - Streaming

Introduction to Spark

Spark Stack - A unified analytics stack



Brief History of Spark

- Originally developed on UC Berkeley AMPLab in 2009.
- Open-sourced in 2010.
- Spark paper came out.
- Spark Streaming was incorporated in 2011.
- Transferred to the Apache Software foundation in 2013
- Spark is a top-level Apache project since 2014
- Spark 2.0 released with major improvements in 2016

Who uses Spark and why?

Data Scientists:

- Analyze and model data.
- Data transformations and prototyping.
- Statistics and Machine Learning.

Software Engineers:

- Data processing systems.
- API for distributed processing dataflow.
- Reliable, high performance and easy to monitor platform.

Spark: Overview

- Spark provides
  - parallel distributed processing,
  - fault tolerance

  On commodity hardware,
- Applications that reuse a working set of data across multiple parallel operations e.g.
  - Iterative Machine learning
  - Interactive Data analysis
- General purpose programming interface
  - Resilient distributed datasets (RDDs)

RDDs - Resilient Distributed Dataset

- Users can
  - Persist the RDDs in Memory
  - Control partitioning
  - Manipulate using rich set of operations
- Coarse-grained transformations
  - Map, Filter, Join, applied to many data items concurrently
  - Keeps the lineage

RDDs

- RDD is represented by a Scala object
- Created in four ways
  - From a file in a shared file system
  - By “parallelizing” an existing scala collection
  - Transforming an existing RDD
  - Changing the persistence of an existing RDD
    - Cache (in-memory, spill otherwise)
    - Save (to HDFS)

RDDs

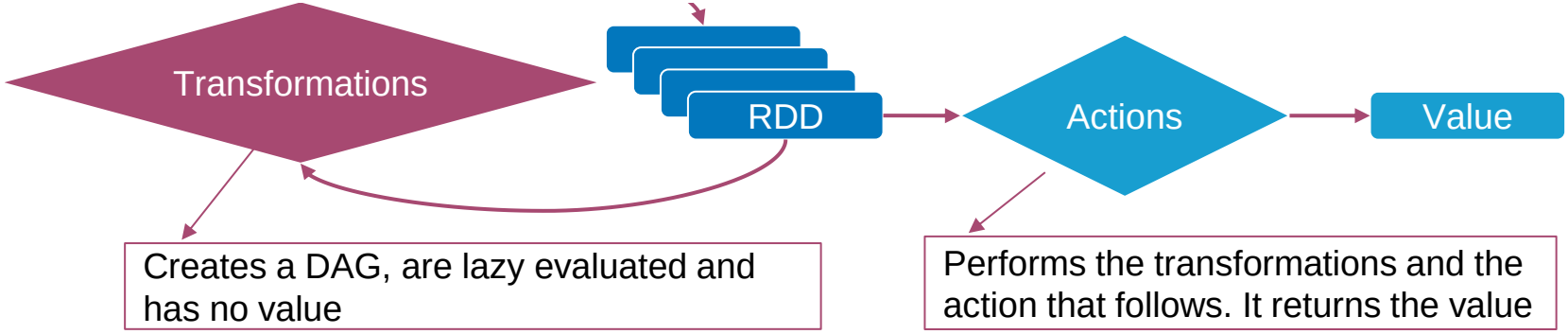
- RDDs can only be created through transformations
  - Immutable,
  - no need of checkpointing
  - only lost partitions need to be recomputed
- Tasks scheduled based on data locality
- Degrade gracefully by spilling to disk
- Not suitable to asynchronous updates to shared state

Parallel Operations on RDDs

- Reduce
  - Combines dataset elements using an associative function to produce a result at the driver program.
- Collect
  - Sends all elements of the dataset to the driver program
- Foreach
  - Passes each element through a user provided function

Resilient **D**istributed **D**ataset (RDD)

- Spark's primarily abstraction.
- Distributed collection of elements, partitioned across the cluster.
  - **Resilient**: recreated, when data in-memory is lost.
  - **Distributed**: partitioned in-memory across the cluster
  - **Dataset**: list of collection or data that comes from file.



Shared Variables

- Broadcast Variables
  - To Share a large read-only piece of data to be used in multiple parallel operations
- Accumulators:
  - These are variables that workers can only “add” to using an associative operation, and that only the driver can read.

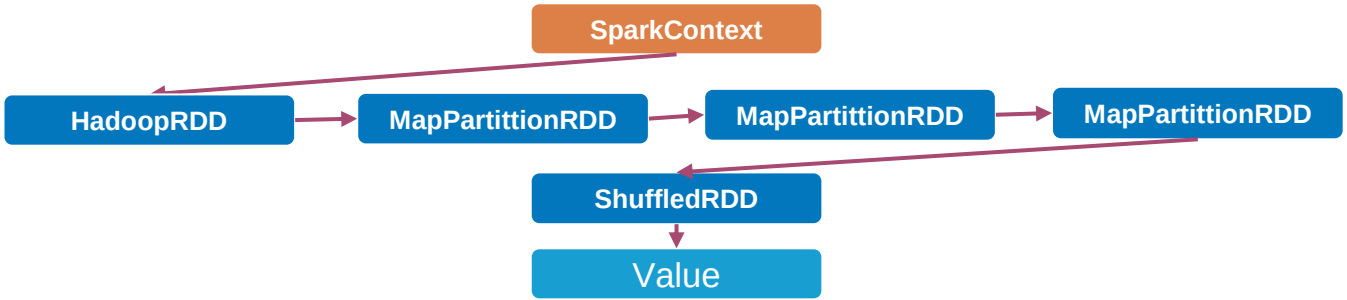
Creating RDDs

- Create some sample data and parallelize by creating and RDD
  - `val data = 1 to 1000`
  - `val distData =sc.parallelize(data)`
- Afterwards, you could perform any additional transformation or action on top of these RDDs:
  - `distData.map { x => ??? }`
  - `distData.filter { x => ??? }`
- An RDD can be created by external dataset as well:
  - `val readmeFile = sc.textFile("README.md")`

RDD Operations

Word Count example

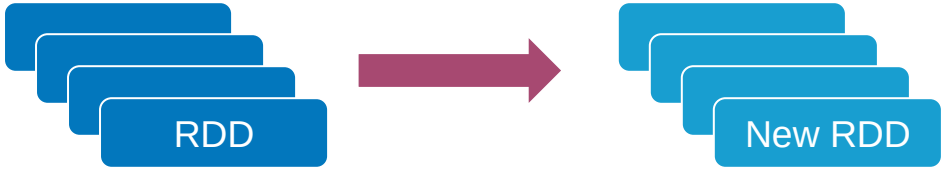
```
val textFile = sparkSession.sparkContext.textFile("hdfs://...")
val wordCounts = textFile.flatMap(line => line.split(" "))
                         .filter(!_.isEmpty())
                         .map(word => (word, 1))
                         .reduceByKey(_ + _) //(a, b) => a + b
wordCounts.take(10)
```



Directed Acyclic Graph (DAG) for Word Count example

RDD Operations

- **Transformations**: Return new RDDs based on existing one (f(RDD) => RDD), e.g filter, map, reduce, groupBy, etc.

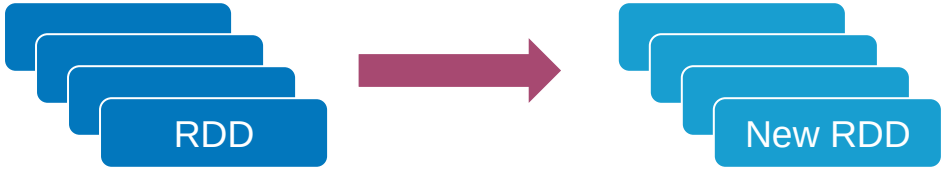


- **Actions**: Computes values, e.g count, sum, collect, take, etc.
  - Either returned or saved to HDFS

RDD Operations

Lazy

- **Transformations**: Return new RDDs based on existing one (f(RDD) => RDD), e.g filter, map, reduce, groupBy, etc.



Eager

- **Actions**: Computes values, e.g count, sum, collect, take, etc.
  - Either returned or saved to HDFS

RDD Transformations (Lazy)

- map:Apply function to each element in the RDD and return an RDD of the result.
- flatMap Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned
- Filter: Apply predicate function to each element in the RDD and return an RDD of elements that have passed the predicate condition, pred.
- distinct:Return RDD with duplicates removed.

RDD Actions (Eager)

- TakeSample
- takeOrdered
- saveAsTextFile
- saveAsSequenceFile

RDD Actions (Eager)

collect: Return all elements from RDD.

count: Return the number of elements

Take(n) Return the first n elements of the RDD.

reduce:Combine the elements in the RDD together using op function and return result.

foreach:Apply function to each element in the RDD

Transformations on Two RDDs (Lazy)

- **union:**Return an RDD containing elements from both RDDs.
- **Intersection:** Return an RDD containing elements only found in both RDDs
- **Subtract:** Return an RDD with the contents of the other RDD removed
- **Cartesian:** Cartesian product with the other RDD.

RDD Operations

Expressive and Rich API

map
filter
groupBy
sort
union
join
leftOuterJoin
rightOuterJoin

reduce
count
fold
reduceByKey
groupByKey cogroup
cross
zip

sample
take
first
partitionBy mapWith
pipe
save
...

RDD Operations

Transformations and actions available on RDDs in Spark.

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] ⇒ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] ⇒ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] ⇒ RDD[U]
	$sample(fraction : \mathrm{Float})$	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	$count()$	:	RDD[T] ⇒ Long
	$collect()$	:	RDD[T] ⇒ Seq[T]
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] ⇒ T
	$lookup(k : \mathrm{K})$	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Pair RDDs

- A common form of data processing
- Main intuition behind the mapreduce
- Often beneficial to project down the complex data types to Key-value pairs

  Distributed key-value pairs
- Additional specialised methods for working with data associated with Keys
- groupByKey(),reduceByKey(),join

Pair RDDs

- Transformations
  - groupByKey
  - reduceByKey
    - Only values of Keys are used for the Grouping
    - More performant
  - mapValues
    - Applies a function to only values in a PairRDD
    - mapValues (def mapValues[U](f: V => U): RDD[(K, U)])
  - keys

Pair RDDs

- Join
  - Inner join, lossy, **only** returns the values whose keys occur in both RDDs
  - leftOuterJoin/rightOuterJoin
- Actions
  - countByKey
    - Counts the number of elements per key , returns a regular map, mapping keys to count

Pair RDDs

Creation: Mostly from existing non-pair RDDs

E.g.

```
val pairRdd = rdd.map(page => (page.title, page.text))
```

- groupByKey
- reduceByKey
- mapValues
- keys
- Join
- left0uterJoin/right0uterJoin
- countByKey

Shuffling

- The shuffle is Spark's mechanism for re-distributing data so that it is grouped differently across partitions.
- This typically involves copying data across executors and machines, making the shuffle a **complex and costly operation.**
- Certain operations within Spark trigger an event known as the shuffle.
- GroupbyKey can cause shuffling

Shuffling

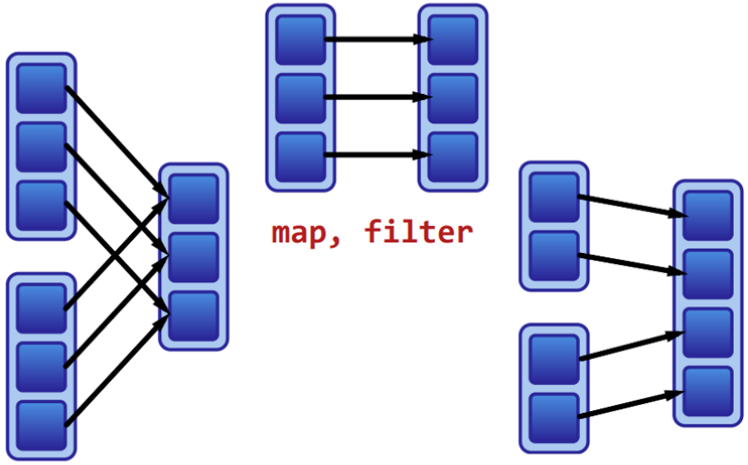
- groupByKey()
  - Shuffles all the keys across network to combine all the keys
- reduceByKey(func: (V, V) => V): RDD[(K, V)J)
  - Conceptually, reduceByKey can be thought of as a combination of first doing groupByKey and then reducing on all the values grouped per key.
  - Reduces on the mapper side first
  - Reduce again after shuffling
  - Less data needs to be sent over the network
  - Non trivial gains in performance

groupByKey and reduceByKey differ in their internal operations

Dependencies / Shuffling

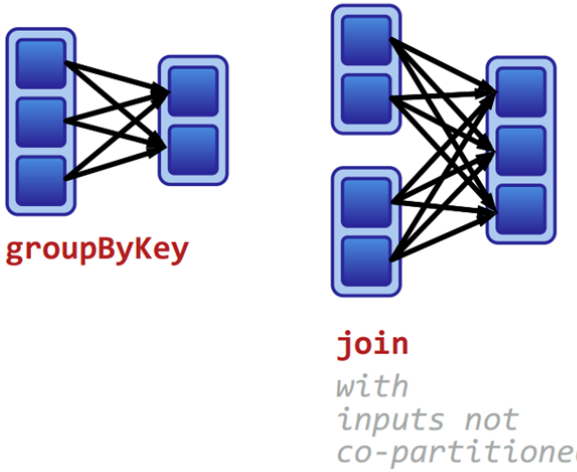
Narrow dependencies:

Each partition of the parent RDD is used by at most one partition of the child RDD.



Wide dependencies:

Each partition of the parent RDD may be depended on by multiple child partitions.



Partitioning

- One partitioning on one machine, or a machine can have many, depending on cores
  - Default = number of cores
- Hash-
  - Hash on key and modulo core size
- Range
  - keys that can have an ordering
- Custom Partitioning , based on keys
    - Only on Pair RDDs

partitionBy

- Range partition
  - Number of partitions
  - PairRdd with ordered keys
- Always **persist**
- Or data will be shuffled in each iteration

Partitioning using transformations

- Partitioner from Parent RDD
  - The RDD that is the result of a transformation on parent RDD usual configured to use the same partitioner as parent
- Automatically set Partitioners
  - e.g. sortByKey uses RangePartitioner
  - groupByKey uses HashPartitioner
- Map and Flatmap loose the partitioner as we can change the key itself
- Use mapValues instead !!

Spark application, configurations, monitoring and tuning

Spark configurations

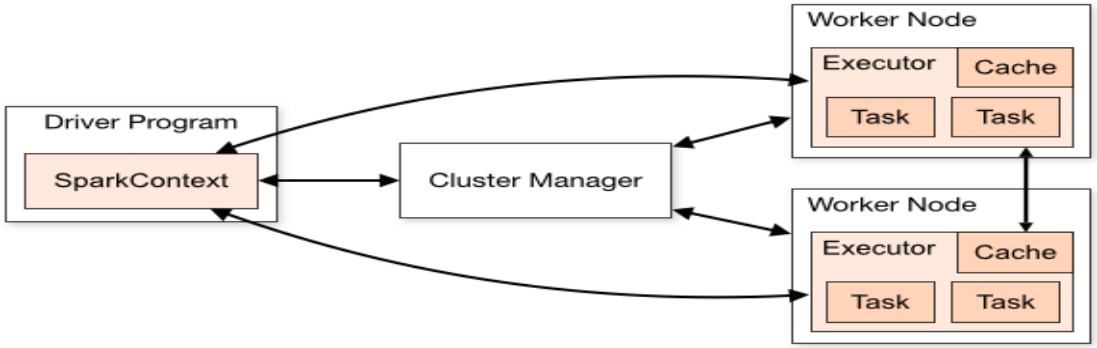
Spark Cluster Overview

Components

Driver aka SparkContext

Cluster Manager (Standalone, Apache Mesos, Hadoop YARN)

Executors



Spark monitoring

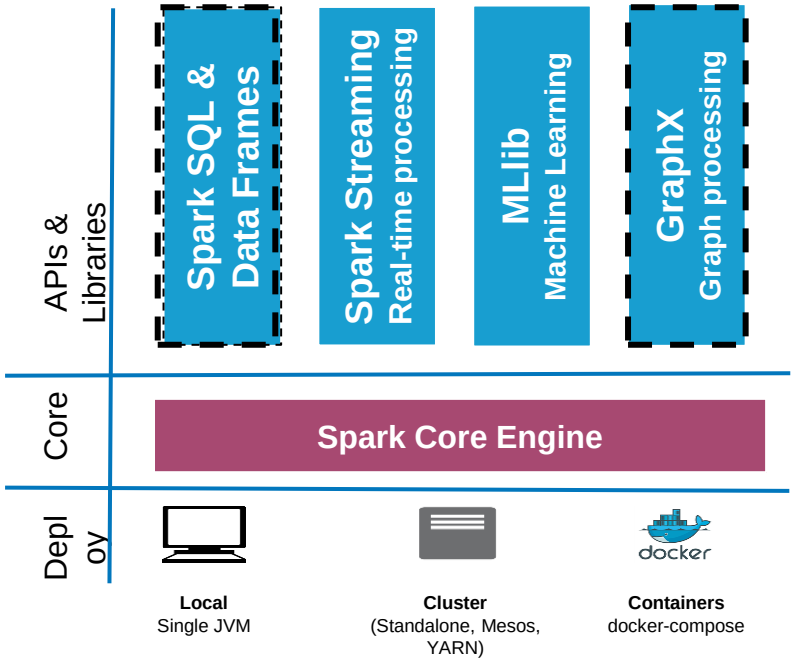
- Web Interfaces
  - **WebUI**
    - Every `SparkContext` launches a web UI, on port 4040, that displays useful information about the application.
  - **Metrics**
    - Spark has a configurable metrics system based on the Dropwizard Metrics Library. This allows users to report Spark metrics to a variety of sinks including HTTP, JMX, and CSV files.
  - **Advanced Instrumentation**
    - Several external tools can be used to help profile the performance of Spark jobs.

Spark tuning

- Tuning Spark
  - **Data Serialization**
    - It plays an important role in the performance of any distributed application.
      - Java serialization
      - Kryo serialization
  - **Memory Tuning**
    - The amount of memory used by your objects (you may want your entire dataset to fit in memory), the cost of accessing those objects, and the overhead of garbage collection (if you have high turnover in terms of objects).
  - **Advanced Instrumentation**
    - Several external tools can be used to help profile the performance of Spark jobs.

Spark Libraries

- A unified analytics stack



Overview

- Spark SQL: Relational Data Processing in Spark
- GraphX: A Resilient Distributed Graph System on Spark

Spark SQL

Motivation

- Support relational processing both within Spark programs
- Provide high performance with established DBMS techniques
- Easily support new data sources, including semi-structured data and external databases amenable to query federation
- Enable extension with advanced analytics algorithms such as graph processing and machine learning

Motivation

- Users:
  - Want to perform ETL-relational processing
    - data Frames
  - Analytics - procedural tasks
    - UDFs

Spark SQL

- A module that integrates relational processing with Spark's Functional programming API
- Spark SQL allows relational processing
- Perform complex analytics
  - Integration between relational and procedural processing through declarative Data Frame
  - Optimizer (catalyst)
    - Composable rules
    - Control code generation
    - Extension points
    - Schema inference for json
    - ML types
    - Query federation

Spark SQL

Three main APIs

- SQL Syntax
- DataFrames
- Datasets

Two specialised backend components

- Catalyst
- Tungsten

Data Frame

- DataFrames are collections of structured records that can be manipulated using Spark's procedural API,
- Supports relational APIs that allow richer optimizations.
- Created directly from Spark's built-in distributed collections of Java/Python objects,
- Enables relational processing in existing Spark Programs
- DataFrame operations in SparkSQL go through a relational optimizer, Catalyst

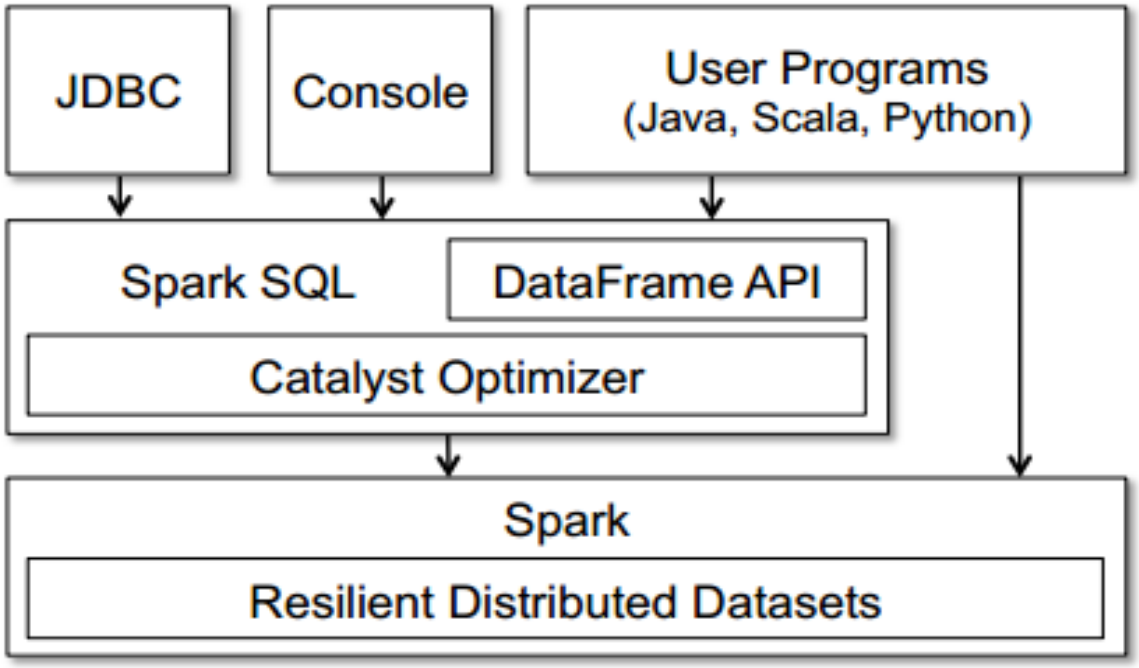
Catalyst

- Catalyst is the first production quality query optimizer built on such functional language.
- It contains an extensible query optimizer
- Catalyst uses features of the Scala programming language,
  - Pattern-matching
  - Express composable rules
  - Turing complete language

Catalyst

- Catalyst can also be
  - extended with new data sources,
  - semi-structured data
    - such as JSON
    - “smart” data stores to use push filters
    - e.g., HBase
    - user-defined functions;
    - User-defined types for domains e.g. machine learning.
- Spark SQL simultaneously makes Spark accessible to more users and improves optimizations

Spark SQL


JDBC
Console
User Programs
(Java, Scala, Python)
Spark SQL
DataFrame API
Catalyst Optimizer
Spark
Resilient Distributed Datasets

DataFrame

- DataFrame is a distributed collection of rows with the “Known” schema like table in a relational database.
- Each DataFrame can also be viewed as an RDD of Row objects, allowing users to call procedural Spark APIs such as map.
- Spark DataFrames are lazy, in that each DataFrame object represents a logical plan to compute a dataset, but no execution occurs until the user calls a special “output operation” such as save

DataFrame

- Created from an RDD using .toDF()
- Reading from a file ()

Example

- ctx = new HiveContext()
- users=ctx.table("users")
- young = users.where(users("age")<21)
- println(young.count())

Data Model

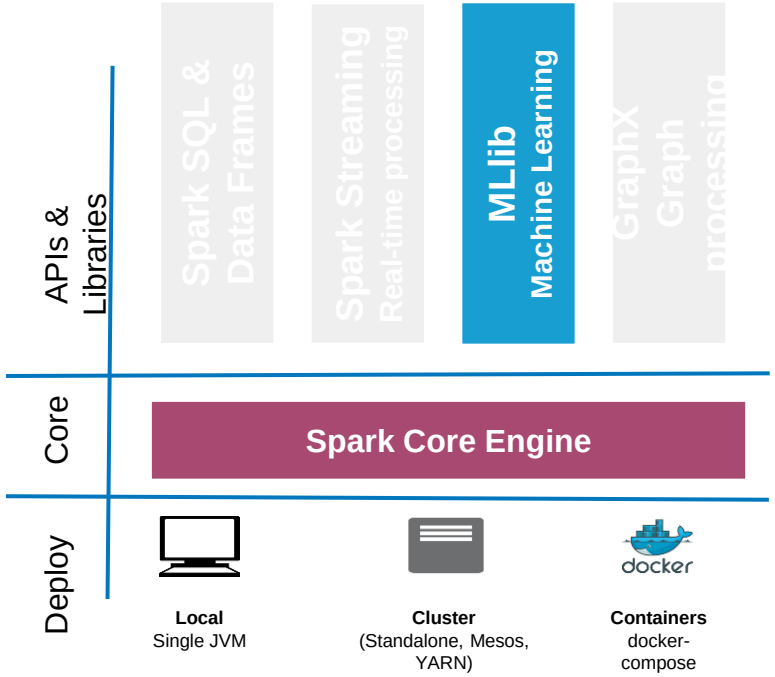
- DataFrames support all common relational operators, including
  - projection (select),
  - filter (where),
  - join, and
  - aggregations (groupBy).
- Users can break up their code into Scala, Java or Python functions that pass DataFrames between them to build a logical plan, and will still benefit from optimizations across the whole plan when they run an output operation.

Optimization

- The API analyze logical plans eagerly
  - identify whether the column names used in expressions exist in the underlying tables,
  - whether the data types are appropriate
- Spark SQL allows users to construct DataFrames directly against RDDs of objects native to the programming language.
- Spark SQL can automatically infer the schema of these objects using reflection

Spark MlLib

Spark ML

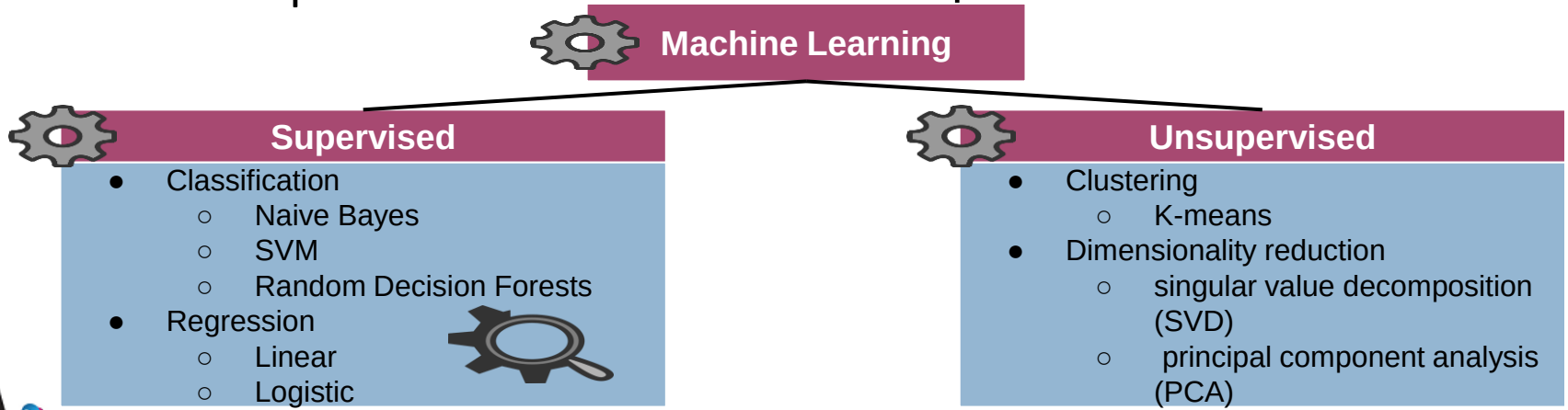

APIs & Libraries
Spark SQL & Data Frames
Spark Streaming Real-time processing
MLlib Machine Learning
GraphX Graph processing
Core
Spark Core Engine
Deploy
Local
Single JVM
Cluster
(Standalone, Mesos, YARN)
Containers
docker-compose
docker

Spark ML

- MLlib is a standard component of Spark providing machine learning primitives on top of Spark
- It provides scalable machine learning, statistics algorithms
- Supports out-of-the-box most popular machine learning algorithms like Linear regression, Logistic regression, Decision Trees
- Is available in Scala, Java, Python, and R

ML Algorithms overview

- Machine learning are separated in two major types of algorithms :
  - Supervised - labeled data in which both, input and output are provided to the algorithm
  - Unsupervised - do not have the outputs in advance



Spark ML-pipelines

- Uniform set of APIs for creating and tuning data processing/machine learning pipelines
- Core concepts:
  - DataFrame: RDD with named columns. SQL-like syntax and other core RDD operations
  - Transformer: DataFrame ⇒ DataFrame. Eg., features to predictions (classifier)
  - Estimator: DataFrame ⇒ Transformer. e.g., learning algorithm
  - Pipeline: Chain of Transformers and Estimators. Specifies the data flow

Spark ML - pipelines: Transformer

- A Transformer transforms one DataFrame to another
- It implements a method `transform()`
- Both feature extractors and (un)trained models are Transformers,
  because they augment input data with features resp. predictions.

Spark ML - pipelines: Estimator

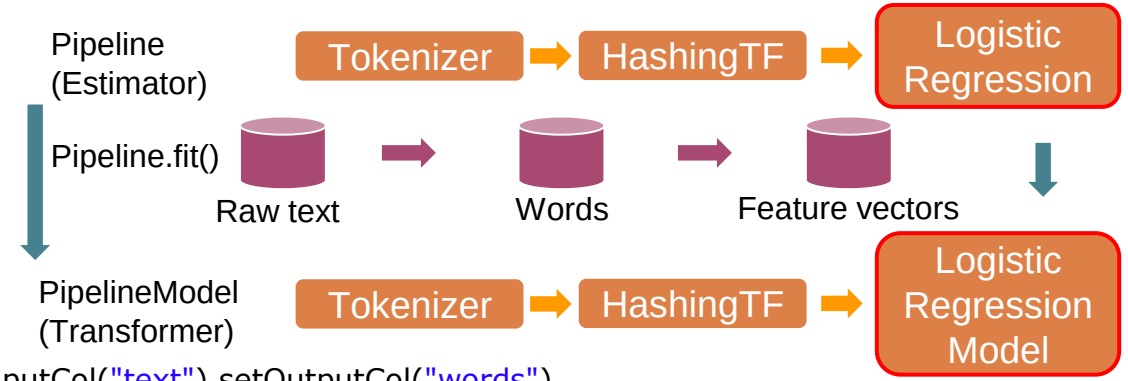
- An Estimator abstraction uses an algorithm which fits a model into a DataFrame
  - Learning algorithms are Estimators
  - Estimators produce models (models are Transformers)
- It implements a method `fit()`

Spark ML - pipelines: Pipeline

- Pipeline: is an Estimator
  - produces a Pipeline-model (a Transformer)
- consists of Transformers and/or Estimators.
- When .fit() is called on the Pipeline:
  - The stages are run in specified order
  - If stage is Transformer, .transform() is called
  - If stage is Estimator, .fit() is called
- pipeline.fit() produces a PipelineModel, where all Estimators are replaced by the Transformers they produced
- ⇒ can easily specify multiple tasks to be trained in a single pipeline and used at test time

Spark ML-pipelines Example

- Split text into words => convert numerical features => generate a prediction model




```
val tokenizer = new Tokenizer().setInputCol("text").setOutputCol("words")
val hashingTF = new HashingTF().setNumFeatures(1000).setInputCol(tokenizer.getOutputCol)
.setOutputCol("features")
val lr = new LogisticRegression().setMaxIter(10).setRegParam(0.01)
val pipeline = new Pipeline().setStages(Array(tokenizer, hashingTF, lr))
val model = pipeline.fit(training.toDF)
val test = sc.parallelize(Seq(
Document(4L, "spark i j k"),
Document(5L, "l m n"),
Document(6L, "mapreduce spark"),
Document(7L, "apache hadoop")))
val predictions = model.transform(test.toDF)
```

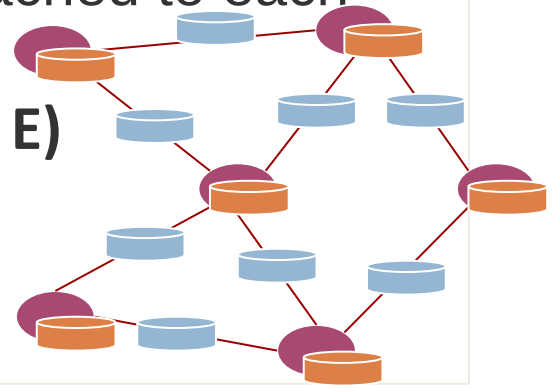
Spark GraphX

Spark GraphX

- Graph computation system which runs in the Spark data-parallel framework.
- GraphX extends Spark's Resilient Distributed Dataset (RDD) abstraction to introduce the Resilient Distributed Graph (RDG)

Spark GraphX

- Spark GraphX - stands for graph processing
  - For graph and graph-parallel computation
- At a high level, GraphX extends the Spark RDD by introducing a new Graph abstraction:
  - a directed multigraph with properties attached to each vertex and edge.
- It is based on Property Graph model → **G(V, E)**
  - Vertex Property
    - Triple details
  - Edge Property
    - Relations
    - Weights



Resilient Distributed Graph (RDG)

- A tabular representation of the efficient vertex-cut partitioning and data-parallel partitioning heuristics
- Supports implementations of the
  - PowerGraph and
  - Pregel graph-parallel
- Preliminary performance comparisons between a popular data-parallel and graph-parallel frameworks running PageRank on a large real-world graph

Graph Parallel

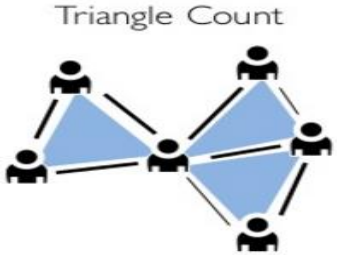
- Graph-parallel computation typically adopts a vertex (and occasionally edge) centric view of computation
- Retaining the **data-parallel metaphor,** program logic in the GraphX system defines transformations on graphs with each operation yielding a new graph
- The core data-structure in the GraphX systems is an immutable graph

GraphX operations

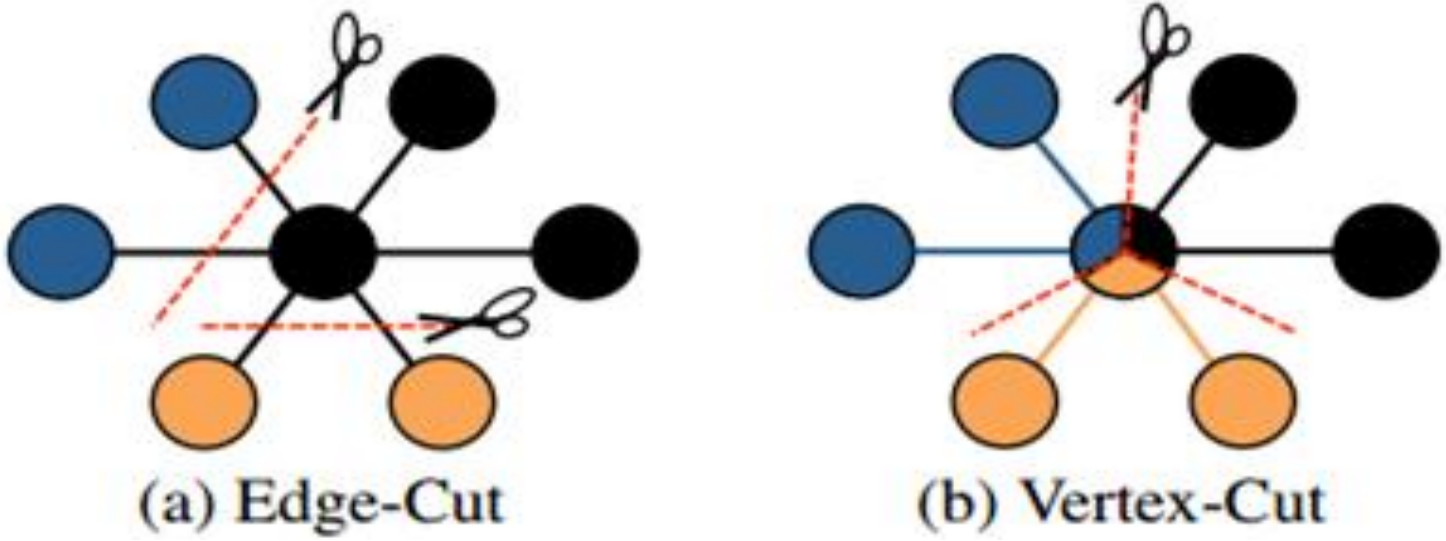
```
class Graph[VD, ED] {
// Information about the Graph
val numEdges: Long
val numVertices: Long
val inDegrees: VertexRDD[Int]
val outDegrees: VertexRDD[Int]
val degrees: VertexRDD[Int]

// Views of the graph as collections
val vertices: VertexRDD[VD]
val edges: EdgeRDD[ED]
val triplets: RDD[EdgeTriplet[VD, ED]]

// Functions for caching graphs
def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD, ED]
def cache(): Graph[VD, ED]
def unpersistVertices(blocking: Boolean = true): Graph[VD, ED]
// Change the partitioning heuristic
def partitionBy(partitionStrategy: PartitionStrategy): Graph[VD, ED]
// Transform vertex and edge attributes
def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
      ----
```

GraphX build-in Graph Algorithms

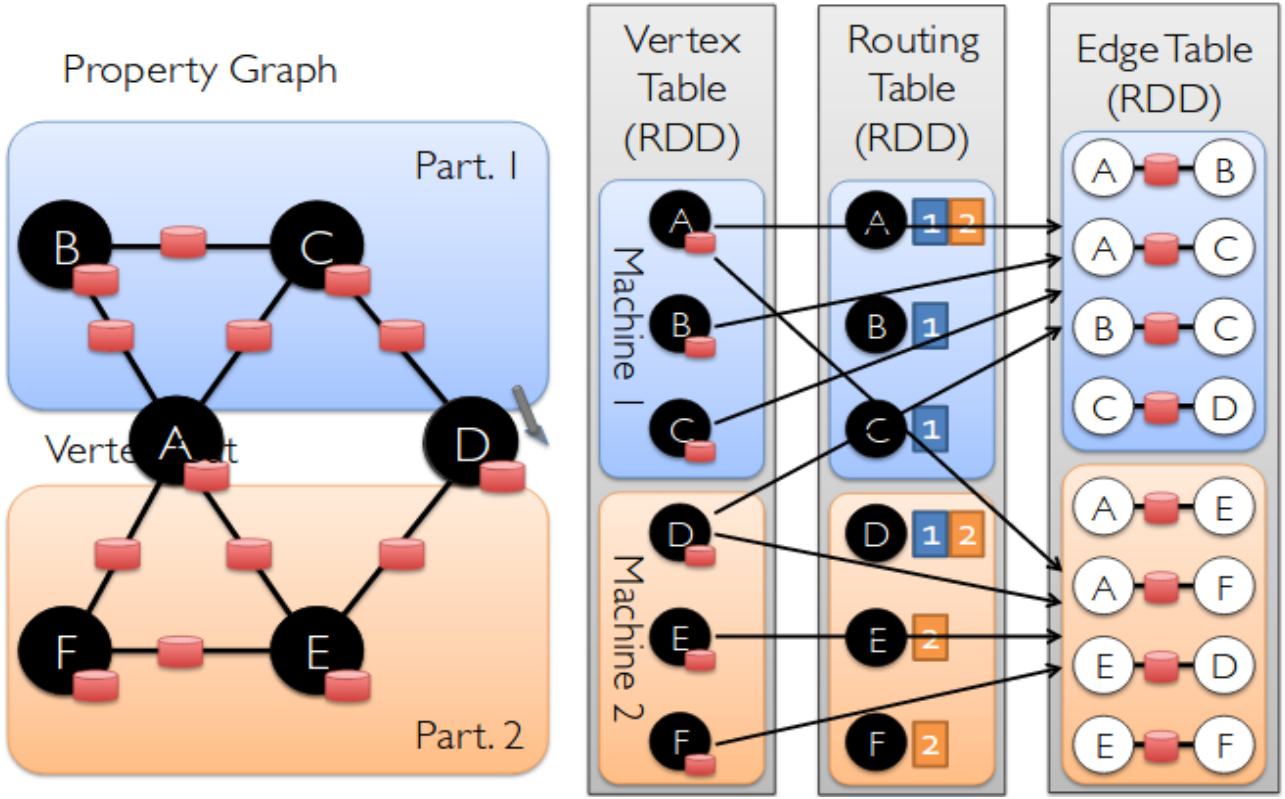
```
// Basic graph algorithms
================================================================
def pageRank(tol: Double, resetProb: Double = 0.15): Graph[Double, Double]
def connectedComponents(): Graph[VertexId, ED]
def triangleCount(): Graph[Int, ED]
def stronglyConnectedComponents(numIter: Int): Graph[VertexId, ED]
```



Edge-Cut vs Vertex-Cut



Encoding Property Graphs as RDDs



Edge Table

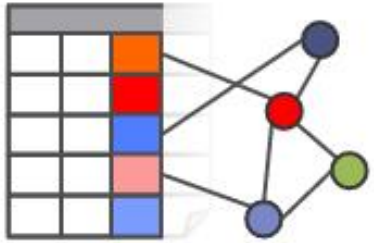
- EdgeTable(pid, src, dst, data): stores the adjacency structure and edge data
- Each edge is represented as a tuple consisting of the
  - source vertex id,
  - destination vertex id,
  - user-defined data
  - virtual partition identifier (pid).

Vertex Data Table

- VertexDataTable(id, data): stores the vertex data, in the form of a vertex (id, data) pairs
- VertexMap(id, pid): provides a mapping from the id of a vertex to the ids of the virtual partitions that contain adjacent edges

New API

Blurs the distinction between Tables and Graphs



New Library

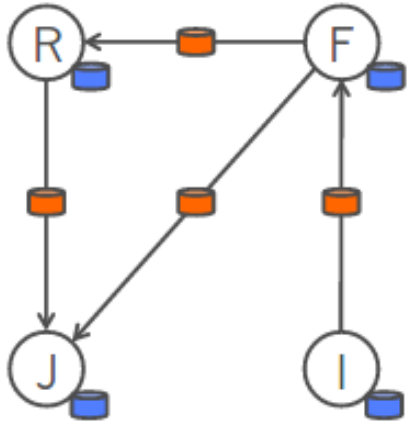
Embeds Graph-Parallel model in Spark

APACHE GIRAPH

Spark

GraphLab

Property Graph



Vertex Table

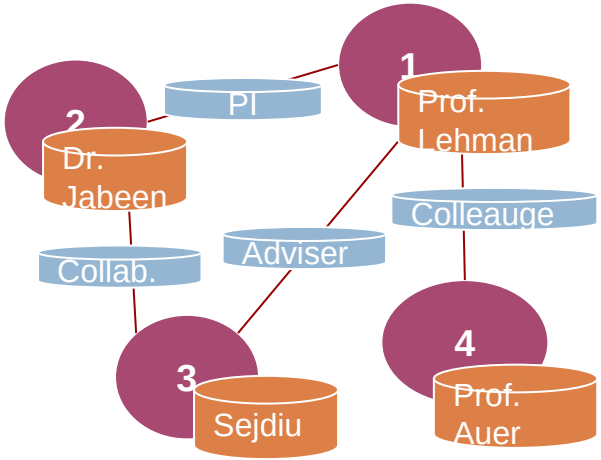
Id	Attribute (V)
Rxin	(Stu., Berk.)
Jegonzal	(PstDoc, Berk.)
Franklin	(Prof., Berk)
Istoica	(Prof., Berk)

Edge Table

SrcId	DstId	Attribute (E)
rxin	jegonzal	Friend
franklin	rxin	Advisor
istoica	franklin	Coworker
franklin	jegonzal	PI

Spark GraphX - Getting Started

❖ Creating a Graph




```
type VertexId = Long
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))]=
spark.sparkContext.parallelize(
                                Array((3L, ("sejdiu", "phd_student")),
                                      (2L, ("jabeen", "postdoc")),
                            (1L, ("lehmann", "prof")),
                                      (4L, ("auer", "prof"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
spark.sparkContext.parallelize(
                                Array(Edge(3L, 2L, "collab"),
                                      Edge(1L, 3L, "advisor"),
                                      Edge(1L, 4L, "colleague"),
                                Edge(2L, 1L, "pi")))
// Build the initial Graph
val graph = Graph(users, relationships)
```

Vertex RDD

vID	Property(V)
1L	(lehmann, prof)
2L	(jabenn, postdoc)
3L	(sejdiu, phd_student)
4L	(auer, prof)

Edge RDD

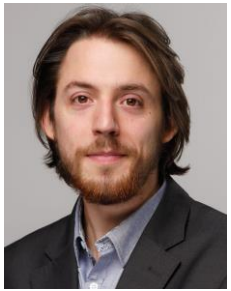
sID	dID	Property(E)
1L	3L	advisor
1L	4L	colleague
2L	1L	pi
3L	2L	collab

GraphX Optimizations

- Mirror Vertices
- Partial materialization
- Incremental view
- Index Scanning for Active Sets
- Local Vertex and Edge Indices
- Index and Routing Table Reuse

References

1. “Spark Programming Guide” - http://spark.apache.org/docs/latest/programming-guide.html
2. “Spark Streaming Programming Guide”- http://spark.apache.org/docs/latest/streaming-programming-guide.html
3. “Spark Cluster Overview” - http://spark.apache.org/docs/latest/cluster-overview.html
4. “Spark Configuration” - http://spark.apache.org/docs/latest/configuration.html
5. “Spark Monitoring” - http://spark.apache.org/docs/latest/monitoring.html
6. “Spark tuning” - http://spark.apache.org/docs/latest/tuning.html
7. MLlib: Machine Learning in Apache Spark by Meng et al. *in Journal of Machine Learning Research 17*, 2016.
8. “Machine Learning Library (MLlib) Guide” - http://spark.apache.org/docs/latest/ml-guide.html
9. Spark SQL: Relational Data Processing in Spark by Armbrust et al. *in SIGMOD Conference,* 2015.
10. “Spark SQL, DataFrames and Datasets Guide” - http://spark.apache.org/docs/latest/sql-programming-guide.html
11. GraphX: Graph Processing in a Distributed Dataflow Framework by Gonzalez et al. *in OSDI,* 2014.
12. “GraphX Programming Guide” - http://spark.apache.org/docs/latest/graphx-programming-guide.html



Dr. Damien Graux
damien.graux@iais.fraunhofer.de

Dr. Hajira Jabeen
jabeen@cs.uni-bonn.de

THANK YOU !

LEARNING, APPLYING, MULTIPLYING BIG DATA ANALYTICS

This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under grant agreement No 809965.